**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе №1**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «Ассоциативный массив»**

Вариант 1

|  |  |  |
| --- | --- | --- |
| Студентка гр. 8301 |  | Чередилина Е.А. |
| Преподаватель |  | Тутуева А. В. |

Санкт-Петербург

2020

# Цель работы

Необходимо ознакомиться со структурой данных «Ассоциативный массив», а также реализовать шаблонный ассоциативный массив (map) на основе красно-черного дерева.

# Постановка задачи

Мне потребовалось реализовать шаблонный ассоциативный массив (map) на основе красно-черного дерева.

Список реализуемых методов:

1. insert (ключ, значение) //добавление элемента с ключом и значением
2. remove (ключ) // удаление элемента дерева по ключу
3. find (ключ) // поиск элемента по ключу
4. clear //очищение ассоциативного массива
5. get\_keys //возвращает список ключей
6. get\_values // возвращает список значений
7. print //вывод в консоль

# Описание программы

Программа представляет собой реализацию структуры данных Map на языке С++. Структура данных Map представляет собой ассоциативный массив, хранящий пары «Ключ-значение». По заданному ключу можно в любой момент времени получить из структуры данных значение, ему соответствующее.

Структура данных Map была реализована на основе красно-чёрного дерева, каждая вершина которого имеет поле ключа и поле значения.

В программе были созданы и использовались следующие классы:

* **Класс List**:

Класс обычного однонаправленного незамкнутого списка, используется для получения списка ключей и списка значений из Map.

* **Класс Map**:

Шаблонный ассоциативный массив, который представляет собой структуру данных Map.

# Оценка временной сложности

## Функция insert (key)

Данная функция имеет временную сложность , так как количество операций не превосходит константу умноженную на высоту дерева, т.е. , где - высота дерева.

## Функция remove (key)

Данная функция имеет временную сложность , так как количество операций не превосходит константу умноженную на высоту дерева.

## Функция find (key)

Данная функция имеет временную сложность , так как количество операций не превосходит константу умноженную на высоту дерева.

## Функция clear

Данная функция имеет временную сложность , так как использует константное число элементарных операций, т.е. количество операций равно количеству вершин.

## Функция get\_keys

Данная функция имеет временную сложность , так как использует константное число элементарных операций.

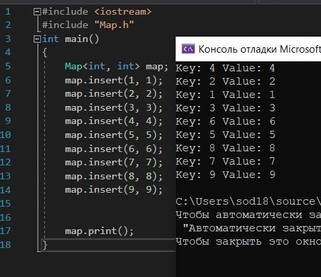
## Функция get\_values

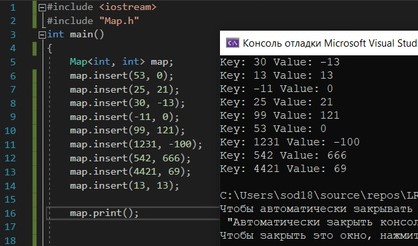
Данная функция имеет временную сложность , так как использует константное число элементарных операций.

## Функция print

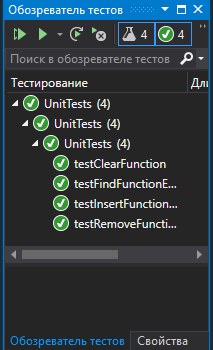
Данная функция имеет временную сложность , так как использует константное число элементарных операций.

# Примеры работы программы





# Примеры работы Unit - тестов



# Листинг

Файл «List.h»

#pragma once

#include<iostream>

using namespace std;

template<typename T>

class List

{

public:

class Node {

public:

Node(T data = T(), Node\* Next = NULL) {

this->data = data;

this->Next = Next;

}

Node\* Next;

T data;

};

public:

void push\_back(T obj) { // add to the end of the list bc

if (head != NULL) {

this->tail->Next = new Node(obj);

tail = tail->Next;

}

else {

this->head = new Node(obj);

this->tail = this->head;

}

Size++;

}

void push\_front(T obj) { // add to top of list bc

if (head != NULL) {

Node\* current = new Node;

current->data = obj;

current->Next = this->head;

this->head = current;

}

else {

this->head = new Node(obj);

}

this->Size++;

}

void pop\_back() { // delete last item bc

if (head != NULL) {

Node\* current = head;

while (current->Next != tail) //looking for the penultimate

current = current->Next;

delete tail;

tail = current;

tail->Next = NULL;

Size--;

}

else throw std::out\_of\_range("out\_of\_range");

}

void pop\_front() { // delete the first item bc-+

if (head != NULL) {

Node\* current = head;

head = head->Next;

delete current;

Size--;

}

else throw std::out\_of\_range("out\_of\_range");

}

void insert(T obj, size\_t k) {// adding an element by index (insertion before an element that was previously available at this index) bc

if (k >= 0 && this->Size > k) {

if (this->head != NULL) {

if (k == 0)

this->push\_front(obj);

else

if (k == this->Size - 1)

this->push\_back(obj);

else

{

Node\* current = new Node;//to add an item

Node\* current1 = head;//to search for the total item

for (int i = 0; i < k - 1; i++) {

current1 = current1->Next;

}

current->data = obj;

current->Next = current1->Next;//transfers the indication to the next element

current1->Next = current;

Size++;

}

}

}

else {

throw std::out\_of\_range("out\_of\_range");

}

}

T at(size\_t k) {//getting an item by index bc

if (this->head != NULL && k >= 0 && k <= this->Size - 1) {

if (k == 0)

return this->head->data;

else

if (k == this->Size - 1)

return this->tail->data;

else

{

Node\* current = head;

for (int i = 0; i < k; i++) {

current = current->Next;

}

return current->data;

}

}

else {

throw std::out\_of\_range("out\_of\_range");

}

}

void remove(int k) { // delete item by index bc

if (head != NULL && k >= 0 && k <= Size - 1) {

if (k == 0) this->pop\_front();

else

if (k == this->Size - 1) this->pop\_back();

else

if (k != 0) {

Node\* current = head;

for (int i = 0; i < k - 1; i++) {//go to the previous item

current = current->Next;

}

Node\* current1 = current->Next;

current->Next = current->Next->Next;

delete current1;

Size--;

}

}

else {

throw std::out\_of\_range("out\_of\_range");

}

}

size\_t get\_size() { //getting list size bc

return Size;

}

void print\_to\_console() { //output of list items to the console through a separator, do not use at bc

if (this->head != NULL) {

Node\* current = head;

for (int i = 0; i < Size; i++) {

cout << current->data << ' ';

current = current->Next;

}

}

}

void clear() { // delete all list items

if (head != NULL) {

Node\* current = head;

while (head != NULL) {

current = current->Next;

delete head;

head = current;

}

Size = 0;

}

}

void set(size\_t k, T obj) // replacement of an element by index with a transmitted element

{

if (this->head != NULL && this->get\_size() >= k && k >= 0) {

Node\* current = head;

for (int i = 0; i < k; i++) {

current = current->Next;

}

current->data = obj;

}

else {

throw std::out\_of\_range("out\_of\_range");

}

}

bool isEmpty() { // check for empty list bc

if (head == NULL) return true;

else return false;

}

public:

List(Node\* head = NULL, Node\* tail = NULL, int Size = 0) :head(head), tail(tail), Size(Size) {}

~List() {

if (head != NULL) {

this->clear();

}

};

private:

Node\* head;

Node\* tail;

int Size;

};

Файл «LR1.cpp»

#include <iostream>

#include "Map.h"

int main()

{

Map<int, int> map;

map.insert(1, 1);

map.insert(2, 2);

map.insert(3, 3);

map.insert(4, 4);

map.insert(5, 5);

map.insert(6, 6);

map.insert(7, 7);

map.insert(8, 8);

map.insert(9, 9);

map.print();

}

Файл «Map.h»

#pragma once

#include <stdexcept>

#include "List.h"

template<typename KeyType, typename ValueType>

class Map

{

public:

enum NodeColor {BLACK, RED};

class Node

{

public:

Node(NodeColor color, KeyType key, ValueType value, Node\* parent = nullptr, Node\* left = nullptr, Node\* right = nullptr):

color(color), key(key), value(value), parent(parent), left(left), right(right)

{

}

NodeColor color;

Node\* left;

Node\* right;

Node\* parent;

KeyType key;

ValueType value;

};

Node\* nullLeaf;

Node\* root = nullptr;

Node\* sibling(Node\* n)

{

if (n == n->parent->left)

return n->parent->right;

else

return n->parent->left;

}

Node\* grandparent(Node\* node)

{

if ((node != nullLeaf) && (node->parent != nullLeaf))

return node->parent->parent;

else

return nullLeaf;

}

Node\* uncle(struct Node\* node)

{

Node\* g = grandparent(node);

if (g == nullLeaf)

return nullLeaf; // No grandparent means no uncle

if (node->parent == g->left)

return g->right;

else

return g->left;

}

void replace\_node(Node\* n, Node\* child) {

child->parent = n->parent;

if (n == n->parent->left) {

n->parent->left = child;

}

else {

n->parent->right = child;

}

}

void delete\_one\_child(Node\* n)

{

/\*

\* Condition: does not have more than one nonzero descendant.

\*/

Node\* child;

if (n->right == nullLeaf) child = n->left;

else child = n->right;

replace\_node(n, child);

if (n->color == BLACK) {

if (child->color == RED)

child->color = BLACK;

else

delete\_case1(child);

}

if (n == root)

{

delete root;

root = child;

}

}

void rotate\_left(Node\* node)

{

Node\* right = node->right;

/\* Create node->right link \*/

node->right = right->left;

if (right->left != nullLeaf)

right->left->parent = node;

/\* Create right->parent link \*/

if (right != nullLeaf)

right->parent = node->parent;

if (node->parent != nullLeaf) {

if (node == node->parent->left)

node->parent->left = right;

else

node->parent->right = right;

}

else {

root = right;

}

right->left = node;

if (node != nullLeaf)

node->parent = right;

}

void rotate\_right(Node\* node)

{

Node\* left = node->left;

/\* Create node->left link \*/

node->left = left->right;

if (left->right != nullLeaf)

left->right->parent = node;

/\* Create left->parent link \*/

if (left != nullLeaf)

left->parent = node->parent;

if (node->parent != nullLeaf) {

if (node == node->parent->right)

node->parent->right = left;

else

node->parent->left = left;

}

else {

root = left;

}

left->right = node;

if (node != nullLeaf)

node->parent = left;

}

void insert\_case1(Node\* n)

{

if (n->parent == nullLeaf)

n->color = BLACK;

else

insert\_case2(n);

}

void insert\_case2(Node\* n)

{

if (n->parent->color == BLACK)

return; /\* Tree is still valid \*/

else

insert\_case3(n);

}

void insert\_case3(Node\* n)

{

Node\* u = uncle(n), \* g;

if ((u != nullLeaf) && (u->color == RED)) {

// && (n->parent->color == RED) Второе условие проверяется в insert\_case2, то есть родитель уже является красным.

n->parent->color = BLACK;

u->color = BLACK;

g = grandparent(n);

g->color = RED;

insert\_case1(g);

}

else {

insert\_case4(n);

}

}

void insert\_case4(Node\* n)

{

Node\* g = grandparent(n);

if ((n == n->parent->right) && (n->parent == g->left)) {

rotate\_left(n->parent);

/\*

\* rotate\_left can be performed as follows, given that there are already \*g = grandparent(n)

\*

\* struct node \*saved\_p=g->left, \*saved\_left\_n=n->left;

\* g->left=n;

\* n->left=saved\_p;

\* saved\_p->right=saved\_left\_n;

\*

\*/

n = n->left;

}

else if ((n == n->parent->left) && (n->parent == g->right)) {

rotate\_right(n->parent);

/\*

\* rotate\_right can be performed as follows, given that there are already \*g = grandparent(n)

\*

\* struct node \*saved\_p=g->right, \*saved\_right\_n=n->right;

\* g->right=n;

\* n->right=saved\_p;

\* saved\_p->left=saved\_right\_n;

\*

\*/

n = n->right;

}

insert\_case5(n);

}

void insert\_case5(Node\* n)

{

Node\* g = grandparent(n);

n->parent->color = BLACK;

g->color = RED;

if ((n == n->parent->left) && (n->parent == g->left)) {

rotate\_right(g);

}

else { /\* (n == n->parent->right) && (n->parent == g->right) \*/

rotate\_left(g);

}

}

void delete\_case1(Node\* n)

{

if (n->parent != nullLeaf)

delete\_case2(n);

}

void delete\_case2(Node\* n)

{

Node\* s = sibling(n);

if (s->color == RED) {

n->parent->color = RED;

s->color = BLACK;

if (n == n->parent->left)

rotate\_left(n->parent);

else

rotate\_right(n->parent);

}

delete\_case3(n);

}

void delete\_case3(Node\* n)

{

Node\* s = sibling(n);

if ((n->parent->color == BLACK) &&

(s->color == BLACK) &&

(s->left->color == BLACK) &&

(s->right->color == BLACK)) {

s->color = RED;

delete\_case1(n->parent);

}

else

delete\_case4(n);

}

void delete\_case4(Node\* n)

{

Node\* s = sibling(n);

if ((n->parent->color == RED) &&

(s->color == BLACK) &&

(s->left->color == BLACK) &&

(s->right->color == BLACK)) {

s->color = RED;

n->parent->color = BLACK;

}

else

delete\_case5(n);

}

void delete\_case5(Node\* n)

{

Node\* s = sibling(n);

if (s->color == BLACK) { /\* this if statement is trivial,

due to case 2 (even though case 2 changed the sibling to a sibling's child,

the sibling's child can't be red, since no red parent can have a red child). \*/

/\* the following statements just force the red to be on the left of the left of the parent,

or right of the right, so case six will rotate correctly. \*/

if ((n == n->parent->left) &&

(s->right->color == BLACK) &&

(s->left->color == RED)) { /\* this last test is trivial too due to cases 2-4. \*/

s->color = RED;

s->left->color = BLACK;

rotate\_right(s);

}

else if ((n == n->parent->right) &&

(s->left->color == BLACK) &&

(s->right->color == RED)) {/\* this last test is trivial too due to cases 2-4. \*/

s->color = RED;

s->right->color = BLACK;

rotate\_left(s);

}

}

delete\_case6(n);

}

void delete\_case6(Node\* n)

{

Node\* s = sibling(n);

s->color = n->parent->color;

n->parent->color = BLACK;

if (n == n->parent->left) {

s->right->color = BLACK;

rotate\_left(n->parent);

}

else {

s->left->color = BLACK;

rotate\_right(n->parent);

}

}

void tree\_clearing(Node\* root)

{

if (root != nullptr && root != nullLeaf)

{

tree\_clearing(root->left);

tree\_clearing(root->right);

delete root;

root = nullptr;

}

if (root == nullLeaf) root = nullptr;

}

void fillListOfValues(Node\* root, List<ValueType>\* list)

{

if (root != nullLeaf && root != nullptr)

{

list->push\_back(root->value);

fillListOfValues(root->left, list);

fillListOfValues(root->right, list);

}

}

void fillListOfKeys(Node\* root, List<KeyType>\* list)

{

if (root != nullLeaf && root != nullptr)

{

list->push\_back(root->key);

fillListOfKeys(root->left, list);

fillListOfKeys(root->right, list);

}

}

void printToConsole(Node\* root)

{

if (root != nullLeaf)

{

std::cout << "Key: " << root->key << " " << "Value: " << root->value << endl;

printToConsole(root->left);

printToConsole(root->right);

}

}

Map()

{

nullLeaf = new Node(BLACK, 0, 0, nullLeaf);

nullLeaf->left = nullptr;

nullLeaf->right = nullptr;

}

void insert(KeyType key, ValueType value)

{

Node\* temporary = root;

Node\* parent = nullLeaf;

if (root == nullptr)

{

root = new Node(BLACK, key, value, parent, nullLeaf, nullLeaf);

return;

}

//Finding the nullLeaf elements that fits our key

while (temporary != nullLeaf)

{

parent = temporary;

if (key < temporary->key)

temporary = temporary->left;

else

temporary = temporary->right;

}

temporary = new Node(RED, key, value, parent, nullLeaf, nullLeaf);

if (key < parent->key)

parent->left = temporary;

else parent->right = temporary;

insert\_case1(temporary);

}

void remove(KeyType key)

{

Node\* temporary = root;

while (temporary != nullLeaf && temporary->key != key)

{

if (temporary->key > key)

temporary = temporary->left;

else temporary = temporary->right;

}

if (temporary == nullLeaf)

throw std::invalid\_argument("No such key in a map");

delete\_one\_child(temporary);

}

ValueType find(KeyType key)

{

Node\* temporary = root;

if (root == nullptr) throw "No such key in a map";

while (temporary != nullLeaf && temporary->key != key)

{

if (temporary->key > key)

temporary = temporary->left;

else temporary = temporary->right;

}

if(temporary == nullLeaf)

throw std::invalid\_argument("No such key in a map");

return temporary->value;

}

void clear()

{

tree\_clearing(root);

root = nullptr;

}

List<KeyType> get\_keys()

{

List<KeyType>\* list = new List<KeyType>();

fillListOfKeys(root, list);

return \*list;

}

List<ValueType> get\_values()

{

List<ValueType>\* list = new List<ValueType>();

fillListOfValues(root, list);

return \*list;

}

void print()

{

if (root == nullptr) throw "Tree is empty";

printToConsole(root);

}

};